Research Proposal

Critical path extraction of distributed applications by just in time kernel tracing

PhD student
Francis Giraldeau

Research Director
Professor Michel Dagenais
École Polytechnique de Montreal

Jury president
Professor Samuel Pierre
École Polytechnique de Montreal

External evaluator
Professor Samar Abdi
Concordia University

Department of Computer Engineering
École Polytechnique de Montreal

ÉCOLE
POLYTECHNIQUE
MONTRÉAL

Winter 2012

Table des matières

Tables index

Figures index

1 Introduction

Large scale cloud-based systems are increasingly prevalent. The complexity of these systems is due to their multicore architectures, virtualization layer, and heterogeneous technologies involved. The configuration space of these systems is thus exponential. The complexity also comes from the constant adaptation of the system to reflect new needs and capacity adaptation. We are interested in the observation of the performance of such systems, with the aim of determining causes of intermittent and difficult to reproduce performance problems. The tools used in industry for the diagnosis of performance problems include resource utilization metrics, profiling tools, network monitoring, layout tools, interactive debuggers and system logs. Information obtained from these different tools is correlated manually to determine the cause of the problem [1]. The ability to observe the execution of such distributed system is a major challenge to master their complexity. In particular, the critical path of execution is essential for understanding the behavior of a distributed application. This path identifies portions of the execution that limit the performance and affect response time.

1.1 Research Question

This research proposal aims to study methods and algorithms to extract the critical path of a distributed application from a kernel trace, while reducing the overhead to a minimum and preserving the temporal properties of the execution. Our challenge is to characterize the time of execution precisely without changing the application source code, considered henceforth as a black box, in order to evaluate the performance of an application and find performance problems rare and difficult to reproduce. To our knowledge, this is a new approach to analyze the performance of distributed applications in a generic way.

The availability of tools based on outcomes of this research could radically improve the way developers and analysts evaluate and find performance problems in distributed systems, and in that respect, represents a significant contribution.

1.2 Objectives

The overall objective of this research is:

> Provide trace analysis algorithms and tools for system administrators and programmers to understand the overall performance of a distributed application.

Specific objectives arising from the overall objective are to :

1. Develop instrumentation and semantics to extract the execution graph of an application from a kernel trace.
2. Extract the execution path of a distributed application online.
3. Determine the critical path of runtime execution.
4. Calculate the resources used in each component for processing.

In summary, the system must profile the execution of the distributed system for the purpose of capacity planning, bottleneck identification and monitoring the system at runtime for isolation of abnormal executions.

For the system to be applicable in practice, the following constraints must be met. The system must :

1. Support distributed systems on a multicore cluster.
2. Have a low overhead for deployment on production systems.
3. Have a low impact on the performance of temporal properties.
4. Be independent of the hardware architecture and transparent to applications.
5. Produce this analysis reliably, even under high load.

The black box approach is considered important to operate with heterogeneous technologies. Operating system tracing meets the requirement for independence of hardware and applications and therefore, is the subject of this research. We believe that kernel tracing has the potential to induce a small overhead during the observation of distributed applications.

1.3 Literature review

This section presents the state of the art in our research field and demonstrates the originality of the contribution described in the next section.

The literature review was conducted using search engines available to the École Polytechnique de Montreal, between January 2011 and February 2012. Articles are mainly journals of IEEE, ACM and Usenix. Google Scholar was used to complete the review.

Concepts and keywords used for the literature review are presented in Table 1.A first search query was made by combining these concepts. We followed references of articles found to be relevant from this first search to ensure we cover all related works. The search targets mainly the last 10 years, but is not limited to this period.

Software	Distributed	Observation	Performance
operating system kernel virtual machine Progra source code binary executable assembly software library	transaction remote procedure Distributed network protocol message passing data center high availability cloud client server	tracing reverse engineering observation Recording benchmarking instrumentation execution graph debugging runtime veritication	critical path scaling queuing model profiling throughtput latency performance counter hardware counter

Table 1: Concepts and search keywords

The inventory includes about 210 items. We note that this is an active area of research by the considerable amount of recent contributions. The volume of publications by year is shown in Figure 1.

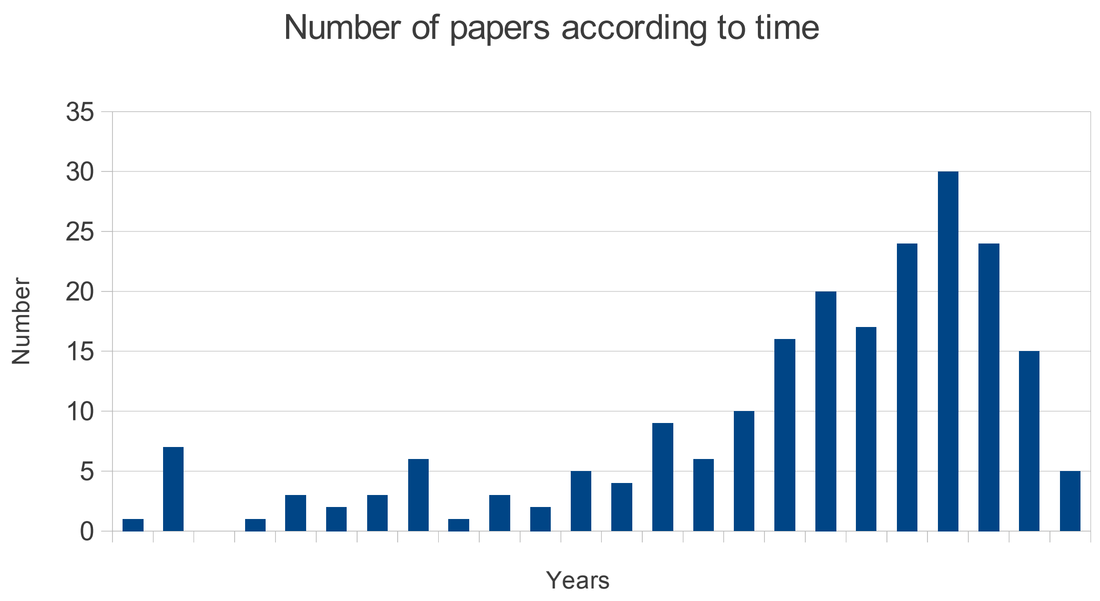


Figure 1: Identified papers according to time

The review begins with the presentation of previous work at the DORSAL laboratory. The aim of this section is to position this proposal in the context of the research group. The next section is about two main categories of distributed analysis of black box systems, namely methods based on statistical correlation of network packets and precise methods of analysis of distributed execution [2]. These two categories are explained and compared in sections 1.3.2 and 1.3.3 respectively. Section 1.3.4 identifies techniques to calculate the critical path of execution, an algorithm essential to our study. The review concludes in Section 1.3.6 with an inventory of tracer systems in order to compare their respective features.

1.3.1 DORSAL Laboratory

The Distributed Open Reliable Systems Analysis Lab (DORSAL) conducted a series of research on distributed multicore systems for critical applications, especially in the field of tracing. Current partners of the laboratory are CAE, the National Defense of Canada, Efficios, Ericsson, Opal-RT and Revolution Linux. The proposed research is built upon earlier work done at the laboratory. The following paragraph summarizes the main contributions of the laboratory.

The laboratory started the Linux Tracing Toolkit (LTT) in 1999 [3], a kernel tracer in the Linux operating system. His successor, LTT Next Generation (LTTng), offers better scaling for multicore architectures through the use of advanced optimizations and lock-free data structures [4–6]. LTTng has been extended to support high performance user mode tracing [7]. Work on time synchronization in distributed traces have been made [8], [9]. LTTng 2.0 was published in March 2012, and represents the state of the art in the field of tracing. This version supports tracing in kernel mode, user mode, dynamic trace points and performance counters. This tracing infrastructure uses per processor, lock-free buffers and records to Common Trace Format (CTF), a standard format tailored to tracing requirements.

From the viewpoint of trace analysis, there are two visualization tools, namely LTT Viewer (LTTV) and the LTTng plugin for Eclipse. These two tools provide a graphical process status according to time, an histogram and basic statistics [10]. Progress has been made towards using traces for security analysis [11], recognizing event sequences using finite state machines [12] and studying the relationship between process blocking [13] and reverse engineering [14]. A hierarchical interval index was developed that allows to recover the state of the system at any time in logarithmic time [15].

1.3.2 Statistical correlation of network packets

Statistical methods found links between events with a temporal correlation technique [16–18]. The method works by recording the time elapsed between incoming and outgoing messages. With an adequate sample of events, the algorithm reports the probability of causality between messages.

Constellation [19] is based on a statistical technique to determine the dependency graph between computers.Network traffic is recorded passively. Addresses and communication ports are used for analysis. The assumption is that a query is probably the cause of another if they are close together in time. By registering a profile over a long period, a statistical test is used to distinguish the related queries. The false positive rate obtained was about 2% for HTTP requests. A similar approach was proposed by Aguilera et al. [20].

Methods based on passive observation of the network have the advantage of not adding additional latency at runtime because the instrumentation is not on the critical path. Also, this approach can be applied to any networked application. Disadvantages are the presence of false positives or relationships that are not identified. In particular, a low accuracy is obtained for rare communication paths. Finally, network observation alone can't determine the links between specific processes involved in the communication, nor links between local processes of a computer, unless additional instrumentation is provided.

1.3.3 Precise methods for distributed execution analysis

This approach requires the knowledge of the event semantic to update a system model. The model and the event must match. The location of the instrumentation changes the nature of available events. Precise methods present a challenge with respect to scaling, because unlike the imprecise techniques that uses sampling, the amount of events to process increases according to the size of the observed system.

The instrumentation of the application or libraries is a simple and effective way to instrument a distributed application accurately [21–32]. The static instrumentation of a program requires changing the source code and adds a dependency on a logging library. There exists dynamic instrumentation methods that directly modify the program in place [33], [34], but such technique vary depending on the programming language and runtime environment. Library instrumentation has the advantage of avoiding the source code modification of applications, but the analysis is restricted to applications using this library, which restricts the set of observable applications. Instrumentation at the application level does not have access to operating system events, such as interrupts and scheduling, limiting the possibility of characterizing the time elapsed at the system level. For the remainder of this section, approaches considered most interesting and relevant topics are detailed.

Google Dapper [21] is based on the instrumentation of the RPC communication library for the analysis of distributed queries. The library assigns a unique identifier to the query at the system entry point. This identifier is then used to correlate all events related to a query. Each RPC request is traced recursively to build the tree of requests according to the components involved. Queries are sampled to minimize the overhead and thereby allowing the scaling to a large number of nodes. A query is recoreded completely or not at all. The time spent in a component is broken down between the communication, waiting, and processing time. Queries are aggregated by similarity to calculate the variance and detect anomalies. This method does not require modification of the application itself, but only applies to applications using the instrumented library, restricting its generality. Moreover, the performance model used is simplistic and specific to RPC applications used in the controlled environment at Google. Finally, the authors ask themselves how to combine the kernel instrumentation in their model.

Jumpshot [22] is a trace visualization tool for MPI applications. The program is instrumented by linking with the library *libmpe*. This library diverts calls to MPI functions, generates events related to communications programs, and then performs the actual call to the function intercepted. The view shows all the processes involved in distributed processing. The temporal view shows the time spent in computation and communication. For each communication, a line shows the relationship between the transmission and reception of a message. Using Jumpshot required to redo the linking of the application, but does not require source code modification. Only MPI applications can be instrumented in this way, reducing the scope of this instrumentation similary to Google Dapper. Moreover, application processing done outside of MPI calls is not visible, preventing the ability to characterize the time elapsed.

Stardust [27] is able to determine the resources used (CPU time, disk access, network transfer and memory usage) for the processing of a distributed query, and the latency induced by each component.In particular, the system produces a very precise analysis by allocating disk writes performed in background to processes cause them. Events are related by a transitive identifier propagated to subsystems. The system was evaluated by the instrumentation of the Ursa Minor NFS server. About 200 tracepoints were added manually in the code to get the result. This is not suitable for observing black box system.

Magpie [29], [30] extracts the execution path of a distributed query. The tracer Event Tracing for Windows (ETW) records events of the operating system, applications and network. The trace analysis is generalized in two ways. First, a schema describes the semantics of events available. Second, the temporal join describes the connection between events to rebuild the request path. The join uses the transitive link between event fields. The transitive closure property forms the complete query. The resulting query includes drives access, scheduling and other events of the operating system. The execution paths are grouped using an algorithm similar to the Levenshtein string distance. The major disadvantage of Magpie is it requires instrumentating the application framework with ETW. This approach requires access to source code of the application framework and its modification, which is not feasible in the case of an heterogeneous infrastructure. Moreover, Magpie was evaluated through the analysis of a Web service, which constitues a subset of applications we want to observe. A similar approach was proposed by Mysore et al. [35].

X-Trace [36] retreives causal communications paths by adding identifier to exchanges packets and messages. This identifier is relayed to the next component or recursively spread to subqueries. The identifier is replicated across the network layers, through the application, TCP, and IP layers. Metadata are stored into option fields of TCP and IP packets. The propagation of metadata to the application layer is protocol dependent. The trace obtained is used to connect the complete causal path accurately through network zones. This technique requires an invasive modification of applications in order to determine the proper propagation of the identifier, and thus is not in itself a method for black box instrumentation. Moreover, no other profiling information is available to analyze the performance of the application.

BorderPatrol [16] retreives the execution path of a distributed application. Calls to the standard C library are intercepted by preloading a shared library. Events are recorded during the establishment of a connection and during the transmission of messages. What distinguishes this study is the utilization of standard protocols to observe and identify the type of messages exchanged between components. The accuracy of the execution path found is based on three assumptions about application behavior, namely the honesty of the application (no bug and no malicious behavior), immediate treatment (without multiplexing) and that processing simultaneous and sequential requests is the same. The application does not need to be modified to be instrumented, which is an advantage. Protocol processors provide a context for the identification of a query. However, the analysis requires the copy of events into an SQL database to perform joins on events. This method is thus neither scalable nor efficient. In addition, the algorithm only works offline. We noticed also that the fixed format of events limits the scope of instrumentation to system calls and communication addresses, which doesn't allow the accurate characterization of processing time.

vPath [37] extracts the execution path of a request by recording events associated to sending and receiving message for each executing thread. Events are recorded by the Xen hypervisor such that no modification to application source code nor inspection of message content are required. The implementation works for the x86 architecture. Each thread is uniquely identified by the tuple consisting of the VM domain and the values of registers CR3 and EBP. The analysis works for applications complying with two programming structures, namely synchronous processing of queries over a reliable communication layer and direct distribution of requests to kernel thread. The study shows that most applications and middlewares follow these common idioms. Causality of the execution is captured by the observation of scheduling events of the operating system, while the causality between the components is determined by exchanged messages. The processing model by a set of worker thread satisfies the hypothesis concerning the required behavior, and this model is prevalent. The system does not support certain pipeline execution that use shared memory because the instrumentation is lacking.

Whodunit [38] extracts the execution of a distributed transaction and produces an execution profile by components. Profiling of each component is obtained with csprof [39]. The instrumentation of messages is performed to determine the links between distributed processes and is implemented by overloading the C library. In order to detect links between local processes, the shared memory communications is traced. This is accomplished by intercepting calls lock functions, then running the protected code block with the QEMU emulator. Addresses involved are recorded to identify producer and consumer processes. Whodunit is specifically designed for multitiers applications.

PreciseTracer [2] extracts the path of a distributed query in a precise and online manner. The activity graph construction of the request is based on the exchange of messages between applications. Each recorded event associates a TCP message to the process involved. These events are then linked together to form the activity graph. Similar graphs are superimposed in order to present a synthetic view of the execution. Two approaches are used to improve the scalability of the analysis. Tracing is enabled globally for a short period, thereby reducing the impact compared to continuously activating the tracing. This decreases the average load over a long periods, but doesn't reduce the extra cost when instrumentation is enabled, which has an impact on requests processed during this time. The other approach consists in a random sampling of events. For a sampling rate of 90% for the RUBiS workload, 90% of the graphs were accurate. The method has the advantage of showing that the algorithm is tolerant to a certain loss rate, but a sampling rate of 90%

decreases only marginally the overhead of tracing and the algorithm has not been tested with a lower sampling rate.

Self-Propelled Instrumentation [40] dynamically instruments a distributed application by following the control flow of execution. A library containing the instrumentation is loaded at runtime and the executable is modified in place to insert jumps to the corresponding instrumentation. The injected share library contains callback functions defined by the user. The instrumentation is propagated to child processes by intercepting fork function calls in the C library. In a similar fashion, when the application connects to a server, it is instrumented in the background before returning. This is done on the peer computer through SSH. We measured that the establishment of a local SSH connection on an Intel 2.5Ghz Core i5 and the OpenSSH server with public key authentication of 2048 bits alone is approximately 250ms. Therefore, this technique does not preserve the temporal properties of the application. Moreover, the propagation of instrumentation has implications for security and system configuration. The client must have the right to change the server executable, which is clearly not desirable in a commercial environment. The client and the server must run on computers of the same architecture, thereby adding a constraint on the supported hardware. Finally, their binary instrumentation only works for compiled executables and excludes bytecode like Java.

1.3.4 Calculating the critical path

Miller et al. [41] present an algorithm for calculating the critical path of a distributed application based on exchanged messages. The Program Activity Graph (PAG) is defined as a directed acyclic graph, where nodes represent the events defining the beginning and end of an activity and whose edges represent the CPU time used. The longest path in this graph represents the critical path. The length of a path is the sum of its segments weights. Their study compares a centralized algorithm to a distributed algorithm for calculating the critical path offline. The focus is on accelerating the offline computation of the critical path by a parallel algorithm.

Hollingsworth et al. [42] propose a method for online calculation of the critical path of an activity graph. The technique avoids both the storing of events and the construstion of the complete graph. It consists in adding the current value of the critical path to outgoing messages. Uppon receiving such message, this value is copied into a local variable. The local processing time is added to the variable. The updated value is transmitted with the reply. Upon receipt of a message containing the CPU usage, the highest value between the local and the received one is retained.

The calculation of the critical path proposed by Miller and Hollingsworth is effective for applications whose performance is CPU bounded. However, this type of analysis is insufficient to determine the critical path of a system that would be bounded by I/O. If the waiting and communication time is included in the weight of edges, then all paths of the graph have the same weight, which makes inoperative existing algorithms for computing the critical path. Also, dependencies between local processes that usin other communication mechanisms such as tubes and signals are not supported, reducing the number of programs that can effectively be observed.

LTTV is a kernel trace viewer developed at the DORSAL laboratory. It contains several modules that display several variables with respect to time, such as the histogram of the number of events, system resources and process status. The analyzer restores the system state from the trace. Regarding the critical path analysis, the dependency module of LTTV [13] uses the kernel tracing to determine wait causes in a program. The analysis focuses on blocking occurring in system calls. The origin of the wakeup event indicates the cause of the wait. The algorithm is offline, and assumes a tree waiting structure that is not appropriate for the general case. This is explained in Section 2.1.

1.3.5 Detection of bottlenecks

Saidi et al. [43] present a method to extract the critical path to determine bottlenecks, whether software or hardware. The method consists in modeling components to be analyzed as automatons. Automaton inputs are recorded events. Links between automatons are defined by the user and form the control flow between processing and waiting. Automata states correspond to the arcs of the dependency graph. These are annotated with the time during which the state is maintained. The system is implemented with a hardware simulator. The low-level hardware analysis is done by instrumenting the simulator with callback functions. The software analysis uses the capability of the simulator to instrument function entry and exit. Automatons are annotated with functions labels. The critical path is calculated without rebuilding the complete graph as described by Hollingsworth et al. The proposed graph visualization method consists to group isomorphic graphs and in annotating edges with the number of executions, the total elapsed time and the proportion of time that belongs to the critical path. The analysis is specifically designed to detect bottlenecks in source code and hardware. In comparison, our method focuses on the general case of the execution at the system level. In addition, the analysis requires knowledge of the application and kernel source code and a hardware model, which is not feasible for black box analysis.

Layered Queuing Network (LQN) has been shown to be effective in determining the capacity and bottlenecks in a distributed system [44]. It has been successfully applied to determine bottlenecks in a soft real-time IP telephony system [45]. This is a generalization of queuing theory for queues in a distributed systems. The average wait time in queue and the average length of the queues are examples of performance measures that the analysis can produce. For some simple models, there is an analytical solution. In other cases, the results can be obtained by discrete event simulation. The difficulty related to the use of LQN consists in the generation of the system performance model. The automation of this approach is proposed to facilitate the construction of the model [46].

Statistical delays of network packets can be used to determine if a bottleneck is reached, without prior knowledge about the maximum capacity of the system [47]. The distribution of the communication latency is compiled for a time window. According to this distribution, the skewness factor of the distribution is calculated. A positive skewness indicates that the distribution is compressed to the limit of the capacity and thus a bottleneck is reached.

1.3.6 Tracers

The LTTng tracer was presented in section 1.3.1. This section describes other available tracers and their features.

Event Tracing for Windows (ETW) [48] is the tracer developed by Microsoft for Windows®. The tracer is able to record events from both user and kernel space. Per CPU buffers are used to reduce synchronization overhead on multicore architectures. The Windows Performance Analyzer (WPA) displays graphs of system resources according to time, calculated from trace events. The analyzer uses the debugging symbols of executables to link metrics to source code.

DTrace [49] is a tracer available for Solaris, FreeBSD and Mach. Features include tracing the kernel, user space programs, registering callback function by event type, event filtering and speculative tracing. Tracing is defined by a script in the D language, which specifies tracepoints to activate and their processing.

SystemTap [50] is a tracer for Linux. It can access tracepoints defined in the kernel, trace user space applications and get performance counters values. A STP script defines tracepoints to activate and specify how to aggregate the data for display. The display function is called periodically, which is the only way to view the data. The STP script is compiled into a kernel module, and then loaded to start the analysis.

Perf [51] is a tracer that includes hardware and software performance counters available from the Linux kernel. It is used to profile the execution of both user and kernel spaces. The availability of performance counter varies depending on the architecture, such as the number of instructions per clock cycle and the number of cache miss. Software performance counter includes events such as minor and major page faults. Data can be stored or displayed in real time.

Ftrace [51] provides details about kernel function calls and the time spent in each. It includes specialized analysis modules, such as latency analysis of the scheduler targeted to real-time tasks.

LTTng will be used to perform this research. This choice is justified because LTTng provides a level of functionality superior to other tracers available for Linux and we have access to source code to perform our experiments. From the standpoint of performance, benchmarks show that the LTTng kernel tracer requires 119 ns to record an event in memory on an Intel Xeon at 2 GHz, under optimal cache conditions [4]. We believe that this performance level is adequate for the project.

1.4 Summary

Approaches based on the framework and application instrumentation enables a detailed analysis of the system behavior in direct relation to the semantic of the application. This relationship is very important to focus optimizations efforts or to narrow down the cause of performance problems. We seek to provide the same benefit without requiring instrumentation of the application framework, libraries or applications.

Of all papers presented in the literature review, the work related to vPath is the one that is most relevant to our goals. The black box approach at the hypervisor meets the criteria for a transparent observation of the execution. This study confirms the feasibility of this method for the cathegory of multitier transactional systems. There are other categories of applications for which the parallel model has not been studied, for instance asynchronous communications. This type of communication is common to increase the parallelism of the execution. However, vPath and the dependency analysis module of LTTV both assumes that waiting is nested between the components. This is too restrictive in the general case.

However, vPath does not link the trace and the source code of the application. In this sense, the integration of statistical profiler in csprof in Whodunit is an interesting approach to this problem because of its precision and low performance impact.

We have not been able to find in the literature a general method for determining online the critical path of distributed execution at the system level with a black box approach. The proposed research focuses on how to extract such a critical path from a kernel trace and look at improving the scalability of this analysis, while maintaining accuracy and generality. We want to continue the work in the direction of dependency analysis module of LTTV at the kernel level. Compared to previous work, the technological context has changed. The LTTng tracing infrastructure now uses lockless Read-Copy Update (RCU) data structures, which reduces contention on shared structures and improves scalability. We have access to a trace index optimized for disk storage, that allows to recover the state of the system at any time. A query on this index has a logarithmic complexity and a constant memory usage. We wish to exploit all information available, including events related to disk requests, memory management, scheduling and interruptions among others, to characterize precisely the execution time of an application at the system level, which would not be possible otherwise. All these elements contribute to the novelty of this research proposal.

Technological and algorithmic contributions are detailed in the next section.

2 Methodology

2.1 Preliminary analysis

Our goal is to reconstruct a directed acyclic graph (DAG) of the runtime execution of an application. Nodes represent application state while edges represent time elapsed between consecutive states. The graph of a process whose execution has no connection with another process degenerates to a list. The graph is used to calculate the critical path of execution.

We saw that annotation of edges with the CPU time proposed by Hollingsworth allows to find the critical path of a CPU bound application. This analysis is suitable for scientific computing, but do not returns the correct critical path for applications in which there is I/O wait. To make the analysis more general, we propose to study events that cause a split and a merge in the graph. These events are more general and include the creation and termination of processes, timers, locks and I/O. An example of such graph is shown in Figure 2. It shows process A that executes a call to clone and creates process B, then waits for the process to exit. The call to clone is treated as a split in the graph, for which two outgoing edges are defined. When the process B exits, an edge connects the end of the process to the wakeup of process A. By this approach, we see that whether the process B uses processor time or not, the execution time of process A depends on the speed of execution of process B. As such, we can conclude that the process B is on the critical path of process A execution.

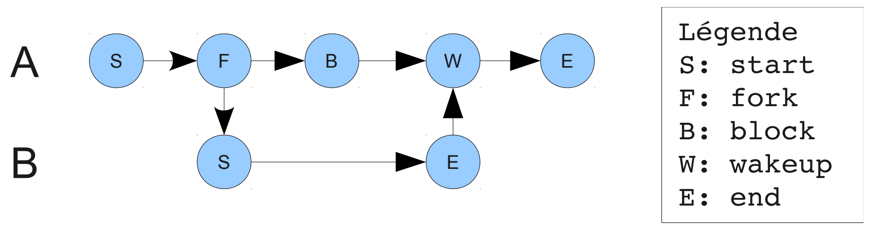


Figure 2: Execution graph involving clone and wait

The principle of operation of the proposed analysis is based on the blocked state of a process and its wakeup signal [13]. An example of this sequence is shown in Figure 3. The state according to time of processes A and B is represented. Process A runs in user space (1) and then executes the system call waitpid (2). If process B has not yet exited, then the process enters blocking state (3), which results in a context switch. When process B exits, the operating system then puts process A in the scheduler queue (4). When process A restarts, the system call returns (5), and the execution in user space is resumed (6). The blocking duration depends on the time taken by process B to complete. We can conclude that a segment of the execution of process B is on the critical path of process A. The wake up signal indicates the wait cause. The calculation of the critical path is performed by traversing the waiting chain.

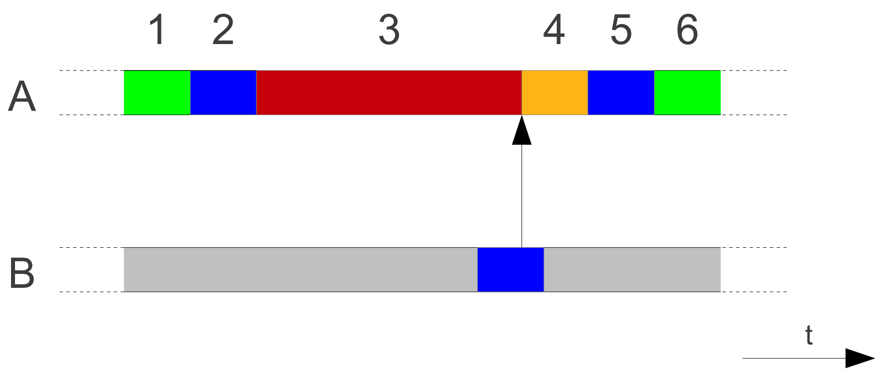


Figure 3: Relationship between two processes

As a preliminary analysis, we studied system calls blocking to assess key behaviors and potential limitations.

- nanosleep : The wake up signal is issued at timer expiration.
- read: the wake up signal is emitted in a softirq to indicate the data is available.

- select : The wake up signal is issued if one or more descriptors are ready, and involved descriptors are identified.
- waitpid : The wake up signal is emitted by the child upon exit.
- futex : The wake up signal is issued by the holder of the lock when released.

For all system calls studied to date, their blocking behavior is the same, and we were able to determine the first degree cause involved. The wake up event indicates control flow change. The control flow is transfered to the process awakened.

The root cause of the wake up is direct in the case of waiting for a process with waitpid. However, the wake up issued from a softirq is indirect. The challenge is to trace the causal chain that led to the execution of the softirq. In most cases, this is an IRQ. Their number indicates the device involved. For example, in the case of an event related to the keyboard, the wait would be assigned to the user. If the interrupt is issued by the disk controller, then it must be possible to trace the chain back to determine the start of the disk request.

We tested the dependency analysis produced by the dependency analysis module of LTTV. The scenario simulates waiting between three processes. The master process performs a system call to clone starting process child 1. The master process continues its execution, then performs the system call waitpid on child 1 still running.Child 1 itself creates child 2 and waits for its termination. The critical path recovered by LTTV is shown in Figure 4 a). This path is incomplete, because if the execution time of child 2 is shortened, then the total execution time of master time would be reduced. Thus, the critical path must go through child 2, as shown in b). The incomplete path found is due to the fact that the algorithm assumes a nested hierarchy of waiting, which is not true in all circumstances. The use of a graph is therefore necessary to represent the dependency chain in the general case.

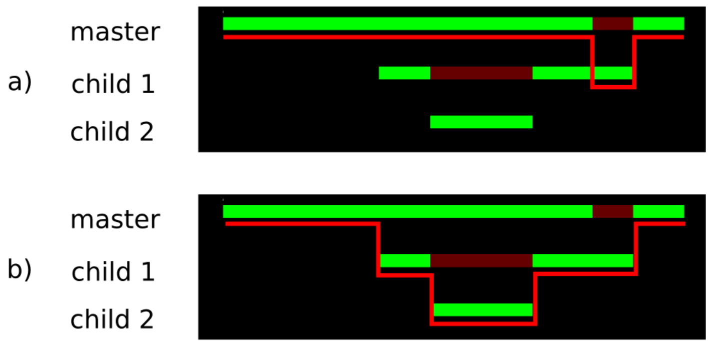


Figure 4: Critical path of the master process

We developed a basic algorithm for extracting the execution path offline and applied it to the trace of the previous example. The resulting graph is shown in Figure 5. It shows states changes from all three processes. In this graph, all paths have the same execution time, so algorithms computing the longest path do not apply for recovering the critical path. To calculate the critical path, the graph is traversed backwards from the end. When a wake up event is encountered, corresponding to a merge, this path is followed. The first split node encountered is followed. This algorithm terminates when the first node of the graph is reached. Visited nodes correspond to the critical path. The critical path of this example is SCSCSEWEWE.

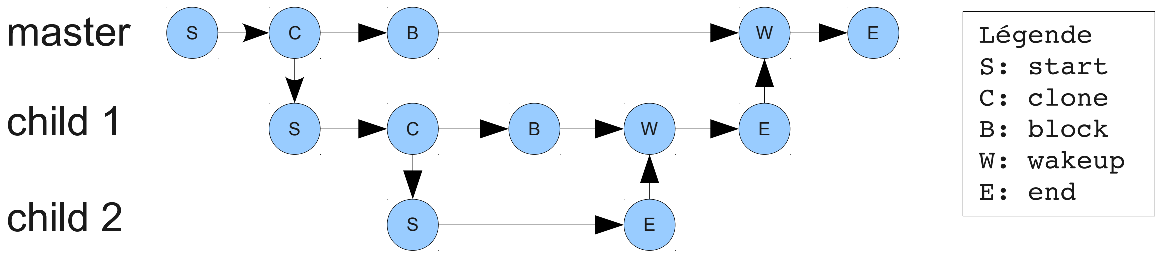


Figure 5: Execution graph of the master process

During the preliminary analysis, we also observed a special case with an OpenMP parallel application. This application simulates an work imbalance problem between threads. By default, OpenMP uses active lock (spin lock) for the syncrhonization barrier. The active wait in the application is not visible from the point of view of the operating system. By setting environment variable OMP_WAIT_POLICY to PASSIVE, blocking is done with a futex system call, and it therefore becomes visible to the operating system. Figure 6 shows the difference between active and passive waiting for the OpenMP application as seen by a kernel trace for the same portion of the program. Thus, for the analysis to be effective, passive waiting is mandatory. Usually, spin lock is reserved for short waiting period, so the impact on the accuracy of results may be low. Similarly, the execution of user space threads and shared memory communications may be problematic because they may be invisible to the operating system. These elements must be taken into account for system evaluation.

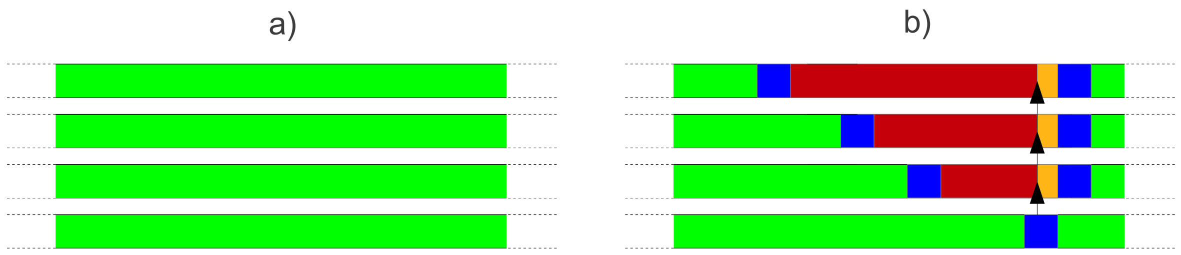


Figure 6: Imbalance occuring in an OpenMP application

Preliminary analysis indicates that our approach has the potential to be generic for a large number of applications and that it helps to find the real critical path of an application not bounded by CPU usage. Limitations concern the synchronization primitives, user space threads and communication that is not visible to the kernel space.

2.2 Execution graph extraction

2.2.1 Semantic

The first step is to develop the full semantic of the execution graph on a local computer. Developing the algorithm on a single trace first simplifies the analysis, because timestamps are globally ordered according to the same global clock. It is expected that the developed algorithm runs offline first. The emphasis is on precision and accuracy of the recovered graph. This algorithm will be the basis for any comparative algorithm developed later.

The current prototype extracts the graph for parent and child relationships. This algorithm must be completed to handle all cases of communication and synchronization between processes. Reading and writing to files, anonymous and named pipes and Unix sockets must be supported. These mechanisms have a similar operation, so their treatment should be related. In addition, locks implemented by passive waiting, visible from kernel space must be handled appropriately.

Waiting for I/O, such as disk access, must be properly identified. To perform this analysis, events related to page fault must be considered. It should be possible to link waitings to disk requests, since it is on the critical path if the application is blocked until the request is done. This situation occurs in case of major page faults. To perform this analysis, the disk request must be followed through all steps until the final wake up signal.

IP sockets are another category of communication mechanism that must be studied, including asynchronous messages. From the perspective of the sender, writing to a socket results in background transmission of packets. On the receiver side, the received packet generates an IRQ, then a softirq that then causes a wake up in the application if it was blocked waiting for the message. The challenge is to follow the communication path through the system call, the network layer and all devices between the sender and the receiver. The full path including low level network events is certainly very precise, but in return requires the tracing of a very large amount of events. An alternative would be to approximate the communication path by observing blocking and wake up from system calls. Both methods should be compared for their overhead and accuracy.

The execution graph can be used as input for further analyses. The dependency tree between processes can be automatically determined by traversing the execution graph.

Figure 7 shows an example of a dependency tree among processes involved in a web browser query.

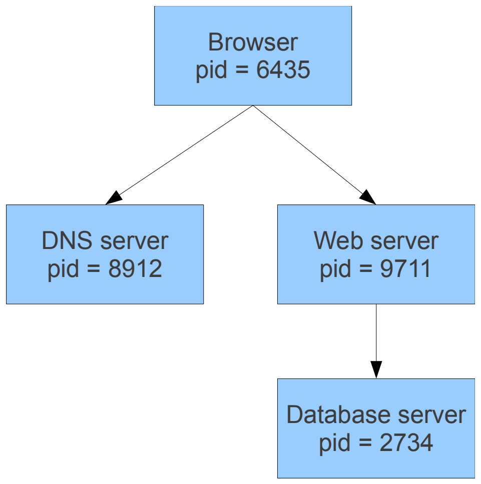


Figure 7: Dependency tree of a web query

The focus will be on the critical path analysis of the execution graph, discussed in Section 2.3.

2.2.2 Trace synchronisation

The algorithm to extract execution graph must be extended to support multiple traces recorded simultaneously on multiple computers with independent clocks. Time synchronization of traces must be considered. To acheive graph consistency, time annotation of edges must be always positive. This requires a partial order on events related to message exchanges. Synchronization based on low level network packets events has been studied previously [8], [9]. The accuracy measured experimentally on a gigabit Ethernet network with the Convex-Hull synchronization algorithm is 30µs. No reversed message were observed in the experiment after synchronization. The use of this algorithm is proposed for synchronizing traces. The correlation between MPI and kernel traces indicates the feasibility of this approach [7].

2.2.3 Aggregation

Concerning the aggregation of distributed traces, we will limit ourselves to a client and server architecture. We know that this centralized architecture limits the scalability of the analysis, but it is sufficient to demonstrate the benefit of our approach. If the infrastructure for control and aggregation based on a tree overlay network becomes available during the project, we could consider contributing a module to improve analysis scalability.

2.2.4 Instrumentation

Additional kernel tracepoints are probably required to perform our analysis. The choice of instrumentation sites and fields to extract must be determined based on analysis requirements, but we must also consider the acceptability of changes we propose for possible inclusion in the Linux kernel. Changes to core kernel algorithms in order to meet tracing needs are unlikely to be adopted. This constraint indicates the need to find identifiers in situ to produce the transitive relation of events. The availability of such consistent identifiers and their accessibility at many levels of the kernel is uncertain.

2.2.5 Just in time tracing

Our goal is to trace a distributed application while involving a minimal overhead. The LTTng kernel tracer logs all enabled events generated globally. This is desirable to recover global system metrics, but only a subset of these events is required to extract the execution graph. We propose just in time kernel tracing to record only relevant events to our analysis in order to drastically reduce the overhead.

Just in time tracing should trace only processes involved in an execution graph from a root process. The functionality required from the viewpoint of the tracer is the activation of tracing for a process set. At the split points of the control flow, the related process is added to the set of traced process. Disabling the tracing for a process can be done when a condition is met, such as closing the file descriptor used for communication.

Tracing activation must also cross machine boundaries. It is proposed to study the propagation using TCP options. If a process is being traces, then an option to outgoing TCP packets could be added that tells the receiver to enable tracing of the process to which the packet is intended. Disabling tracing could occur when receiving a packet for which the option is not set.

Tracing activation must be optimized to have minimal impact on process latency. Usually, tracepoints are enabled before starting the tracing and the impact of the dynamic activation has not been studied.

2.2.6 Sampling

Sampling is another technique proposed to reduce the tracing overhead. Random sampling of events is not suitable for our analysis, because the loss of a single event can affect the accuracy of the extracted graph.Rather than sampling individual events, we intend to apply sampling on complete execution graph. On a loaded system, a random wake up signal sent to the process under observation activates tracing. The tracing would propagate to other processes involved as discussed above. The request would be followed for the processing duration.

2.2.7 Speculative tracing

Speculative tracing with DTrace is available [49]. It allows continuous recording of events in memory, but saving on disk is done only if a condition is satisfied. Unlike the dynamic activation, tracepoints are always enabled, which implies an additional constant cost to trace in memory, but reduces the size of the resulting trace on disk. The use of speculative tracing could be used to record only the blocking system calls, the major page faults and the corresponding wake up events. System calls and page faults would all be traced, but recorded only if a context switch related to a passive waiting occurs.

2.2.8 Online processing

We wish to extend the algorithms and tools to make them work online. The online mode would produce the analysis incrementally as events are available. Online processing requires the adjustment of several algorithms : the time synchronization, the construction of the graph and the computation of the critical path (discussed in Section 2.3) must all be done incrementally. Recent advances to be published about online incremental synchronization of traces will be useful for the implementation of the system.

2.3 Calculating the critical path

The calculation of the critical path takes as input the execution graph and returns the subgraph limiting the execution time. The algorithm performing the critical path computation was presented briefly in Section 2.1. This algorithm has the advantage of being deterministic and of having an execution time proportional to the length of the critical path. Because it works backwards, the algorithm can begin when the complete graph is available.

We propose to compare backward and forward traversal of the graph. The advantage of forward computation is to build the graph and compute the critical path while reading the trace. This is suitable for online processing. However, it requires breadth-first exploration of the complete graph and doing so may implies high memory requirements.

It is necessary to read all events of the trace to find the status of each process in time. It imply that the processing time is proportional to the size of the trace, and it is the best performance acheivable without pretreatment. We wish to study the use of indexes to reduce the computational complexity of graph extraction. These indexes are advantageous because the cost of indexing can be spread over several subsequent analyzes. Indexing can be done online and thus reduce the analysis time perceived by the user.

For example, the indexing of certain types of events of the trace would allow to read only portions that are useful for the graph construction by skipping events that are irrelevant. The acceleration produced would be proportional to the number of unread events.

The interval history tree [15] allows to find the complete state of the system at a specific point in logarithmic time.Index build time is proportional to the size of the trace and uses constant memory. The index has been successfully tested for traces up to 1TB. This index allows iterate on changes of an attribute. For example, an attribute may be the state of a process. Intervals have a beginning and a duration, which allows to iterate directly on state changes rather than the events of the trace. The advantage for the calculation of the execution graph is to navigate through blocked states of processes, because they indicate control flow change. The interval history tree has the potential to reduce the computational complexity in proportion to the number of blocks rather than the number of events. The performance gain is expected to be related to the ratio of events compared to the number of intervals. This ratio depends on the characteristics of the application and thus must be obtained experimentally.

2.4 Tracing guarantees

The accrurary of precise method is vulnerable to event loss. In the case where events are lost, the graph could be wrong and render the whole analysis useless. Generating events in memory exceeding the disk (or network) bandwidth available to save events can cause buffer overrun. In this situation, older events not yet consumed are overwritten by newer events. This design limits the perturbation of the traced system.

The problem of event loss can be addressed from two different angles. The first is to develop an algorithm that has some tolerance to losses. This solution has been studied in PreciseTracer [2]. Tolerance to losses contributes to the robustness of the algorithm and could nevertheless produce approximate results. This tolerance is generally desirable and should be evaluated. The alternative is to sacrifice performance for accuracy by forcing the system to record all events generated in all circumstances.

Ensuring no event loss is non-trivial in the case of kernel tracing, because the operating system generates events and can not be stopped during writing on disk. Also, event production is governed by a burst stochastic process. These are possible solutions envisaged.

- Dynamically allocate buffers : Simplifies the procedure for determining the appropriate buffer size required to handle bursts of events.If the steady state event production rate exceeds the disk bandwidth, this method is not effective in guaranteeing no losses.
- Dynamically disable events : Lower priority events are disabled when the capacity of the buffer exceeds a certain threshold, thus the production rate of events would be decreased. This method does not guarantee all events are saved, but prioritizes losses.
- Prioritize the consumer daemon : If the daemon priority is increased and it does not yield the processor, then it may limit the event production rate from other processes on the system.Events related to scheduling and interruptions will still be issued. This method would ensure that no event is lost if the bandwidth is greater than the rate of event production related to the synchronization itself.
- Use flight recording mode : In this mode, events are recorded continuously in the circular buffer in memory until a condition is satisfied.At that time, tracing is stopped and buffers are written to disk. This technique ensures that the latest events will be written without loss. However, the duration of the trace saved depends on the average event production rate and buffer size.

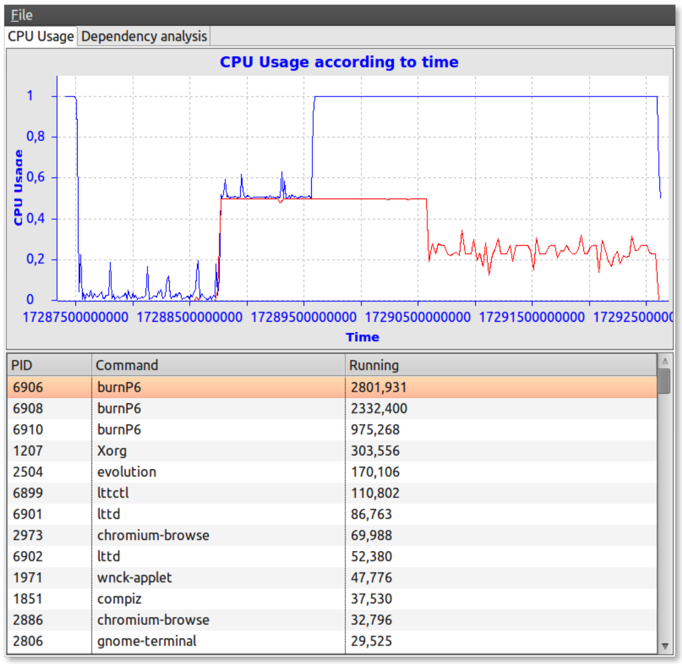
We plan to develop heuristic combining different approaches in order to increase robustness. An analytical model of the system would be useful to simulate the performance of different heuristics quickly, and then to implement the solution providing the lowest loss rate for evaluation in extreme conditions.

2.5 Performance metrics

System metrics are useful to determine the cost of a distributed execution. We identified the kernel instrumentation to extract system metrics in [52]. Main metrics are CPU usage, memory allocation, disk and network I/O. We developed a prototype to show feasibility of this approach for the CPU usage. A screenshot of the prototype is shown in Figure 8. The example shows the start of three processes consisting of a busy loop on a dual-processor machine. The contention appears after starting the third process. The processor utilization rate is recovered from scheduling events. Unlike the proc interface to read the average values of CPU usage periodically, this method can detect a very short duration process. We also note an increased accuracy, because the period over which the average is calculated is customizable. By defining a null average period, the square wave representing context switches would be retrieved from the kernel trace, a precision that is not achievable by polling.

In addition to these metrics, it is possible to attach the value of processor performance counters to kernel trace events. This feature would facilitate the correlation of information between layers of the system, while enjoying the low performance impact incurred by performance counters.

We wish to extend the algorithms for calculating metrics to accumulate all resources used in the execution graph. Graph edges would be annotated with resources used during this interval. The total cost of the execution could then be computed with high precision.



PID	Command	Running
6906	burnP6	2801,931
6908	burnP6	2332,400
6910	burnP6	975,268
1207	Xorg	303,556
2504	evolution	170,106
6899	lttctl	110,802
6901	lttd	86,763
2973	chromium-browse	69,988
6902	lttd	52,380
1971	wnck-applet	47,776
1851	compiz	37,530
2886	chromium-browse	32,796
2806	gnome-terminal	29,525

Figure 8: CPU usage recovered from kernel trace

2.6 Evaluation

The LTTng tracing infrastructure and the CTF trace format will be used for experimental development, as previously discussed. The availability of the open source code allows us to modify the software in order to implement algorithms and perform measurements.

Two aspects are considered for the evaluation of graph extraction algorithms, namely the overhead and accuracy. Our evaluation strategy is inspired by vPath [37]. Techniques will be tailored to kernel traces.

To assess the accuracy of the algorithms, we need programs with known behavior. To this end, we started the project workload-kit, which contains a set of programs producing a known load on the system. For example, we built an algorithm using a calibration step to compute the counter value required to keep busy the processor for a certain amount of time, regardless of the processor frequency. These utilities will be extended to simulate all basic situations the algorithm must supports. A utility will be developed to generate arbitrary distributed loads. This utility will record its own execution graph, as a baseline for comparing the graph found from the kernel trace.

As our algorithms depend on the characteristics of the network, we will study the extraction of the graph by varying the latency. This test should make visible any blocking that may occur in an application and allow to study the behavior when kernel network buffers are full.

Tools developed will be validated on real applications too, not only with test programs. RUBiS [53] is a web service in Java replicating the functionality of the auction site eBay. It is possible to simulate the load of an arbitrary number of concurrent clients. We intend to make the tracing of the web client and the infrastructure to demonstrate the proper functioning of our method. We will verify results by inspection, while the overhead will be measured experimentally according to the number of simultaneous clients.

We will study the robustness of the tracer to avoid event loss by producing an extreme load on the system. We will experimentally determine the combined load producing the greatest rate of events. Tracing will be performed under these conditions, then the number of lost events will be observed. We will also observe the behavior by varying the bandwith of the device used for trace recording. The analytical model will be validated with empirical observations.

To conduct the assessment of online algorithms, we plan to use the cluster of 32 processors of the DORSAL laboratory. The accuracy of online results will be compared to the offline algorithm. The extra cost will be measured according to the frequency of updates.

We will use developed tools to solve performance problems faced by our industrial partners. We want to show the usefulness of tools in real situations, for which they have improved the resolution of performance problems on complex distributed systems.

2.7 Additionnal work

Within the project, we expect some developments not associated with a scientific contribution, but nevertheless necessary. For example, we want to develop a graphical interface to display the execution graph and its critical path. This GUI will be developed in Java as an Eclipse plugin. This interface will be published as a standalone application whose name is LTTng Studio.

3 Planning

The course of the PhD is planned for 3.5 years. During this period, the publication of three journal articles is planned. This section details the expected contributions and the completion schedule.

3.1 Contributions

Programs will be published in order to get feedback from our industrial partners. Contributions to the following software components are expected :

- Linux kernel : Additional Instrumentation.
- LTTng : Just in time tracing, tracing guarantees.
- Workload-kit : Test programs.
- LTTng Studio : Algorithms for analysis and visualization.

The first step consist in the offline extraction of execution graph, which is prior to the the computation of the critical path and online algorithm. Contributions concerning the robustness and optimizations are planned at the end, as they aim to improve results obtained previously. Specific contribution from the last section will be adjusted according to the project's progress and ongoing research opportunities. Contribution alternatives are presented.

The rest of the section details the scope of planned papers and journals targeted for publication are listed. Finally, relevant conferences for dissemination of results are presented.

3.1.1 Scientific papers

Here is the list of papers expected as part of the research.

Article 1 : Dependency analysis from kernel trace

Summary : With the increasing complexity of IT systems in the cloud, new tools are required to observe precisely the runtime execution of applications for debugging or profiling purposes. We propose a new method to observe the runtime behavior of distributed applications from a kernel trace. Tools work with any recent Linux kernel, without modifications to applications nor libraries, and work accros machine boundaries. The method is based on blockings that can occur inside system calls to infer change in the control flow. The method extracts execution graph with low runtime overhead and compute the cost of the whole execution. We present LTTng Studio, a tool to visualize the distributed execution of an application. Our results are presented in terms of accuracy, runtime overhead and algorithmic complexity. We used LTTng Studio to observe a set of basic distributed system primitives and the end-to-end performance of the RUBiS three tier auction web site.

Article 2 : Online critical path extraction of distributed applications from kernel trace

Summary : Characterizing with precision the time spent in each component of a heterogenous distributed application, in response to a request, is a challenging task. Yet, the response time is an important metric for user satisfaction. We present a novel technique to analyse the critical path of a distributed application at the operating system level. Other techniques were able to compute the critical path in terms of CPU usage. While it is effective for most scientific computations, we show that fails for mixed CPU and I/O workloads of typical web transaction systems. We address this limitation by following control flow changes uppon blocking and wakeup of distributed processes. We compare offline and online algorithms to compute the critical path. We evaluate the capability of the tool to recover critical path of parallel applications and a three tier transaction system, in terms of precision and runtime overhead.

Article 3 a) : Just in time kernel tracing

Summary : We demonstrated previously a method based on kernel tracing for critical path recovery of distributed applications. We propose just in time tracing to improve drastically the scalability of the analysis. This technique records the minimal set of events required to recover the execution graph. It works by activating the instrumentation by following the control flow of the application at runtime, across machine boundaries. To increase scalability even more, we propose to use a sampling technique that records only a subset of all executions. The sampling can be adjusted to provide desired maximum overhead. We use a transaction web site representative of commercial workloads to compare accuracy and overhead between complete and just in time tracing.

Article 3 b) : Preventing loss of operating system event trace

Summary : Kernel tracing is a precise and rich source of runtime information. Precise analysis performed on kernel events are subject to fail in case of event loss. For example, if a scheduling event is lost on a CPU, then following events are accounted to the wrong process. Event loss occurs when circular buffers are full and generated events overrides ones that are not already consumed. This situation has a higher probability to occur under abnormal load, exactly when tracing would be usefull to understand the problem. In the traditionnal producer and consumer problem, the producer blocks until some buffer space is freed. In the case of kernel tracing, blocking the kernel to flush buffers would result in an immediate deadlock of the system. Increasing buffer size can handle event bursts, but the average event rate must always be lower than the disk or network bandwith. The analysis is complexified by the presence of a positive feedback loop in the system, in which the consumer activity is recorded and generates events, that may lead to instability. We developed a model of event production and consumption of kernel tracing to study this problem. We describe the model validation emperically according to a set of workloads. We evaluate the effectiveness and cost of various heuristics in terms of their ability to prevent event loss.

3.1.2 Selected journals

We target three journals with peer review for publication. The impact factor is obtained from the ISI Web Of Knowledge database[1].

ACM Transactions on Computer Systems (TOCS)

Description : This journal is dedicated to research on computers and accepts articles about operating systems, performance models and performance analysis, which is within the scope of all our contributions.Impact Factor: 1.889

IEEE Transactions on Computers

Description : Monthly publication covering inter alia operating systems and performance analysis.Founded in 1953, in 2004 it was the 16th most cited paper in the field of electrical and electronic engineering. Topics covered by this journal are directly related to our contributions. Impact Factor: 1.608

IEEE Transactions on Parallel an d Distributed Systems

Description : Monthly publication about techniques for performance measurement on clusters of computers, operating systems concepts, performance models and simulation of multiprocessor systems. The distributed aspect of our method and the ability to study parallel applications are relevant to this journal. Impact factor : 1.575

3.1.3 Conferences

Here is the list of five relevant conferences for the dissemination of our research.

1 ISI Web of Knowledge – Journal Citation Reports® http://admin-apps.webofknowledge.com

ACM SIGOPS EuroSys

Description : EuroSys is the European Chapter of ACM SIGOPS interest group on operating systems, multiprocessor architectures and computer modeling.EuroSys conference is held annually in the spring.

ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)

Description : Topics of this conference that are relevant to our research are verification and testing of distributed systems, multiprocessor architectures and parallel programming.PODC conference is held annually in the summer.

ACM Symposium on Operating Systems Principles (SOSP)

Description : The conference covers topics related to operating systems and distributed systems.It is mentioned that outstanding papers may be referred to the ACM Transactions on Computer Systems journal for possible publication. The conference is held annually in the fall.

Runtime Verification

Description : The conference addressed runtime observation of systems and associated analyzes.Aspects relevant to our research programs are instrumentation, tracing and visualization of program execution. The conference is held annually in the fall.

USENIX Symposium on Operating Systems Design and Implementation (OSDI)

Description : Conference organized by The Advanced Computing Systems Association on similar topics that ACM SOPS. The conference is held annually in the fall.

IEEE International Symposium on Performance Analysis of Systems and Software

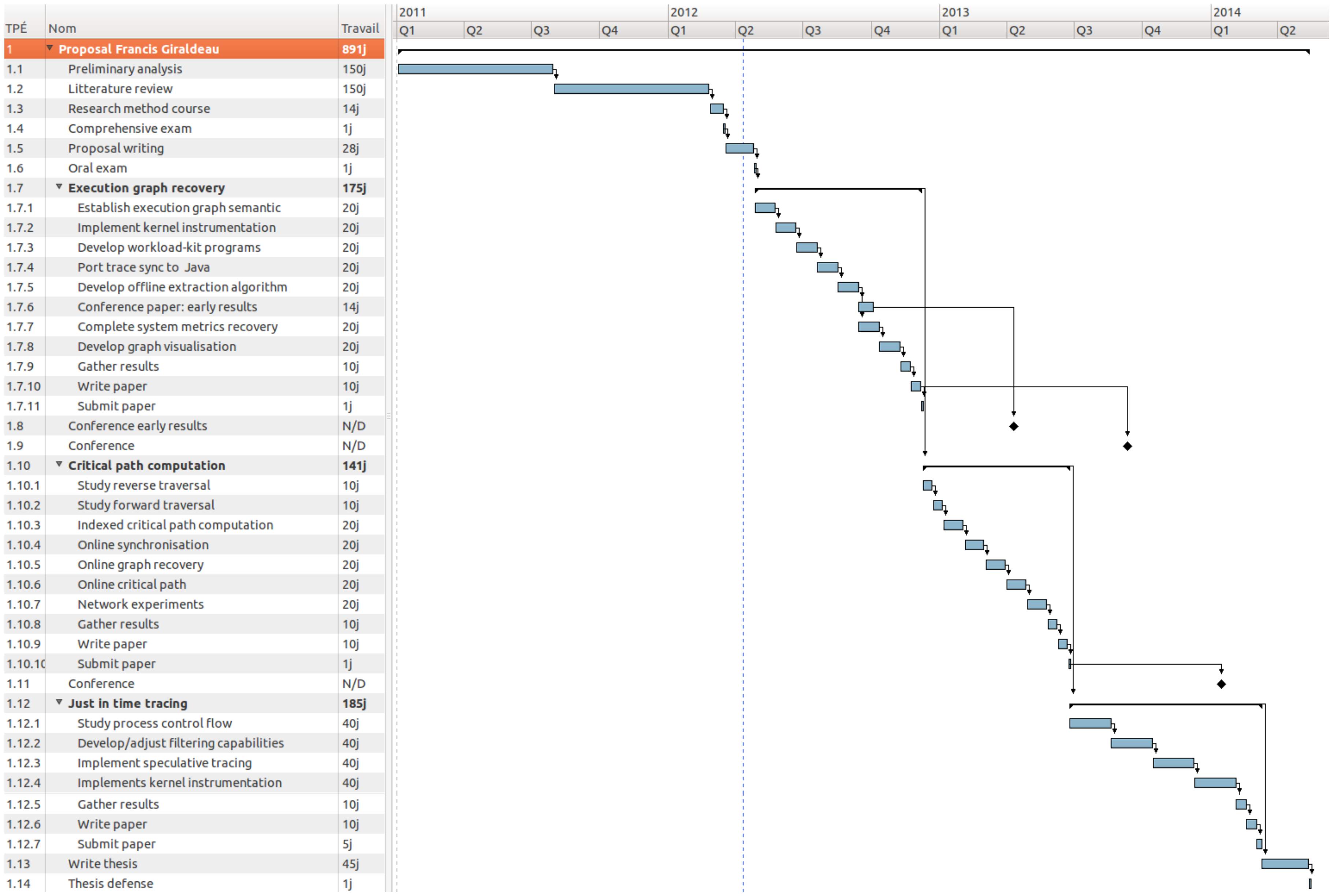
Description : The conference specifies accepting articles in the field of tracing and performance analysis, especially for distributed systems. The conference takes place in spring.

3.2 Schedule

The project schedule is shown in Figure 9. The project began in January 2011. During this period, the literature review and prototyping has been made. This work led to writting this proposal. All academic courses required will be completed in April 2012.

An internship at Ericsson headquarters in Sweden in the Component Based Architecture (CBA) group is planned for the summer term 2012. The planned activities will be conducted on site.

The main steps in the realization of each phases are documented. Regarding the last contribution, only one of the alternatives is presented. If the last contribution changes, the due date remains the same.



TPÉ	Nom	Travail
1	Proposal Francis Giraldeau	891j
1.1	Preliminary analysis	150j
1.2	Litterature review	150j
1.3	Research method course	14j
1.4	Comprehensive exam	1j
1.5	Proposal writing	28j
1.6	Oral exam	1j
1.7	**Execution graph recovery**	**175j**
1.7.1	Establish execution graph semantic	20j
1.7.2	Implement kernel instrumentation	20j
1.7.3	Develop workload-kit programs	20j
1.7.4	Port trace sync to Java	20j
1.7.5	Develop offline extraction algorithm	20j
1.7.6	Conference paper: early results	14j
1.7.7	Complete system metrics recovery	20j
1.7.8	Develop graph visualisation	20j
1.7.9	Gather results	10j
1.7.10	Write paper	10j
1.7.11	Submit paper	1j
1.8	Conference early results	N/D
1.9	Conference	N/D
1.10	**Critical path computation**	**141j**
1.10.1	Study reverse traversal	10j
1.10.2	Study forward traversal	10j
1.10.3	Indexed critical path computation	20j
1.10.4	Online synchronisation	20j
1.10.5	Online graph recovery	20j
1.10.6	Online critical path	20j
1.10.7	Network experiments	20j
1.10.8	Gather results	10j
1.10.9	Write paper	10j
1.10.1(	Submit paper	1j
1.11	Conference	N/D
1.12	**Just in time tracing**	**185j**
1.12.1	Study process control flow	40j
1.12.2	Develop/adjust filtering capabilities	40j
1.12.3	Implement speculative tracing	40j
1.12.4	Implements kernel instrumentation	40j
1.12.5	Gather results	10j
1.12.6	Write paper	10j
1.12.7	Submit paper	5j
1.13	Write thesis	45j
1.14	Thesis defense	1j

Figure 9: Research gantt chart

4 Future work

Some relationships between processes are not visible from the kernel space, including spin lock, communication through shared memory and user space threads. These limitations may affect the accuracy of the analysis for certain applications. The easy solution would be to instrument user space libraries. However, the challenge is greater to observe these primitive programming transparently, without modification to the libraries, and involves trade-offs in terms of performance and accuracy. In the case of shared memory, it may be instrumented by protecting virtual memory pages, however, interruptions raised at each memory access is very expensive. It would thus be interesting to find a more efficient method. With regard to the execution of threads in user space, it would be appropriate to check the possibility of using the base pointer register to detect changes of user space thread. The effectiveness of this technique depends on the conventions used by such libraries and the efficiency to detect the change of the base pointer register. Finally, spin lock may be identifiable by establishing their execution profile obtained by performance counters during contention, but this is a statistical approach for which there is uncertainty.

The establishment of an LQN model from the perspective of the operating system could determine bottlenecks in a distributed system in a generic way. The model would be powered by kernel trace events and provide continuous performance metrics for its supervision, or used as part of a quality plan.

Finally, we planned to demonstrate our system with Linux by accepting the assumption that this operating system is constant. In reality, there are a variety of operating system in use. As kernel tracing is available in other operating systems, it would be appropriate to study the compatibility of event semantic and to abstract it in order to perform the analysis independently of the operating system. Because there is no standard to our knowledge in this field, the difficulty toward achieving this abstraction is due to the likely impedance mismatch between instrumentation of different operating systems.

5 Conclusion

Kernel tracing provides detailed and precise information that allow understanding the behavior of a system at runtime. Kernel tracing is non-invasive, has low overhead and provides an overview of the system. This research proposal aims to continue the efforts of the DORSAL laboratory in the field of kernel tracing by developing algorithms and tools for the extraction of execution graph of a distributed application by calculating the critical path. We believe that this original analysis will help its users understand how the execution time is divided between distributed components. This task is impractical to do so generically with other methods, or even by manually analyse traces obtained on complex systems. Furthermore, the concept of just in time tracing has the potential to reduce the analysis overhead by many orders of magnitude, thereby making possible its continuous activation on production systems. These analyses will change the way we see distributed systems in the future and make kernel tracing tools essential for computer engineering.

6 References

[1] R. Barrett, E. Kandogan, P. P. Maglio, E. M. Haber, L. A. Takayama, and M. Prabaker, “Field studies of computer system administrators: analysis of system management tools and practices,” in *Proceedings of the 2004 ACM conference on Computer supported cooperative work*, 2004, pp. 388–395.

[2] B. Sang, J. Zhan, G. Lu, H. Wang, D. Xu, L. Wang, and Z. Zhang, “Precise, scalable, and online request tracing for multi-tier services of black boxes,” *IEEE Transactions on Parallel and Distributed Systems*, no. 99, p. 1–1, 2010.

[3] M. R. Dagenais, K. Yaghmour, C. Levert, and M. Pourzandi, “Software Performance Analysis,” *Arxiv preprint cs/0507073*, 2005.

[4] M. Desnoyers, “Low-impact operating system tracing,” Thesis, École Polytechnique de Montréal, Montréal, 2009.

[5] M. Desnoyers and M. R. Dagenais, “The lttng tracer: A low impact performance and behavior monitor for gnu/linux,” in *Proceedings of the Ottawa Linux Symposium*, 2006, vol. 2006.

[6] M. Desnoyers and M. Dagenais, “Low disturbance embedded system tracing with linux trace toolkit next generation,” in *ELC (Embedded Linux Conference)*, 2006.

[7] B. Poirier, R. Roy, and M. Dagenais, “Unified Kernel and User Space Distributed Tracing for Message Passing Analysis,” in *Proceedings of the First International Conference on Parallel, Distributed and Grid Computing for Engineering*, 2009, pp. 218–234.

[8] E. Clément and M. Dagenais, “Traces synchronization in distributed networks,” *Journal of Computer Systems, Networks, and Communications*, p. 5, 2009.

[9] B. Poirier, R. Roy, and M. Dagenais, “Accurate offline synchronization of distributed traces using kernel-level events,” *ACM SIGOPS Operating Systems Review*, vol. 44, no. 3, pp. 75–87, 2010.

[10] D. Toupin, “Using tracing to diagnose or monitor systems,” *Software, IEEE*, vol. 28, no. 1, pp. 87–91, 2011.

[11] A. S. Sendi, M. Jabbarifar, M. Shajari, and M. Dagenais, “FEMRA: Fuzzy Expert Model for Risk Assessment,” 2010, pp. 48–53.

[12] G. Matni and M. Dagenais, “Automata-based approach for kernel trace analysis,” in *Canadian Conference on Electrical and Computer Engineering (CCECE)*, 2009, pp. 970–973.

[13] P. M. Fournier and M. R. Dagenais, “Analyzing blocking to debug performance problems on multi-core systems,” *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 77–87, 2010.

[14] M. Desnoyers and M. Dagenais, “OS tracing for hardware, driver and binary reverse engineering in Linux,” *CodeBreakers Journal*, vol. 1, no. 2, 2006.

[15] A. Montplaisir-Goncaves, “Stockage sur disque pour accès rapide d’attributs avec intervalles de temps,” Mémoire, École Polytechnique de Montréal, Montréal, 2011.

[16] E. Koskinen and J. Jannotti, “Borderpatrol: isolating events for black-box tracing,” in *ACM SIGOPS Operating Systems Review*, 2008, vol. 42, pp. 191–203.

[17] P. Reynolds, J. L. Wiener, J. C. Mogul, M. K. Aguilera, and A. Vahdat, “WAP5: black-box performance debugging for wide-area systems,” in *Proceedings of the 15th international conference on World Wide Web*, 2006, pp. 347–356.

[18] S. Agarwala, F. Alegre, K. Schwan, and J. Mehalingham, “E2eprof: Automated end-to-end performance management for enterprise systems,” in *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2007, pp. 749–758.

[19] P. Barham, R. Black, M. Goldszmidt, R. Isaacs, J. MacCormick, R. Mortier, and A. Simma, “Constellation: automated discovery of service and host dependencies in networked systems,” *TechReport, MSR-TR-2008-67*, 2008.

[20] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen, “Performance debugging for distributed systems of black boxes,” in *ACM SIGOPS Operating Systems Review*, 2003, vol. 37, pp. 74–89.

[21] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag, “Dapper, a large-scale distributed systems tracing infrastructure,” *Google Research*, 2010.

[22] O. Zaki, E. Lusk, W. Gropp, and D. Swider, “Toward scalable performance visualization with Jumpshot,” *International Journal of High Performance Computing Applications*, vol. 13, no. 3, pp. 277–288, 1999.

[23] M. Casas, R. Badia, and J. Labarta, “Automatic analysis of speedup of MPI applications,” in *Proceedings of the 22nd annual international conference on Supercomputing*, 2008, pp. 349–358.

[24] W. E. Nagel, A. Arnold, M. Weber, H. C. Hoppe, and K. Solchenbach, *VAMPIR: Visualization and analysis of MPI resources*. Citeseer, 1996.

[25] D. Gunter, B. Tierney, B. Crowley, M. Holding, and J. Lee, “Netlogger: A toolkit for distributed system performance analysis,” in *Proceedings of 8th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, 2000, pp. 267–273.

[26] B. Tierney, W. Johnston, B. Crowley, G. Hoo, C. Brooks, and D. Gunter, “The NetLogger methodology for high performance distributed systems performance analysis,” in *Proceedings of the Seventh International Symposium on High Performance Distributed Computing*, 1998, pp. 260–267.

[27] E. Thereska, B. Salmon, J. Strunk, M. Wachs, M. Abd-El-Malek, J. Lopez, and G. R. Ganger, “Stardust: tracking activity in a distributed storage system,” in *ACM SIGMETRICS Performance Evaluation Review*, 2006, vol. 34, pp. 3–14.

[28] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer, “Pinpoint: Problem determination in large, dynamic internet services,” in *Proceedings of International Conference on Dependable Systems and Networks*, 2002, pp. 595–604.

[29] R. Isaacs, P. Barham, J. Bulpin, R. Mortier, and D. Narayanan, "Request extraction in Magpie: events, schemas and temporal joins," in *Proceedings of the 11th workshop on ACM SIGOPS European workshop*, 2004, p. 17.

[30] P. Barham, R. Isaacs, R. Mortier, and D. Narayanan, "Magpie: Online modelling and performance-aware systems," in *Proceedings of the 9th conference on Hot Topics in Operating Systems-Volume 9*, 2003, p. 15–15.

[31] M. Chen, E. Kiciman, A. Accardi, A. Fox, and E. Brewer, "Using runtime paths for macroanalysis," in *Proceedings of the 9th conference on Hot Topics in Operating Systems-Volume 9*, 2003, p. 14–14.

[32] B. P. Miller, M. Clark, J. Hollingsworth, S. Kierstead, S. S. Lim, and T. Torzewski, "IPS-2: The second generation of a parallel program measurement system," *IEEE Transactions on Parallel and Distributed Systems*, vol. 1, no. 2, pp. 206–217, 1990.

[33] C. K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *ACM SIGPLAN Notices*, 2005, vol. 40, pp. 190–200.

[34] S. Chiba, "Javassist: Java bytecode engineering made simple," *Java Developer's Journal*, vol. 9, no. 1, 2004.

[35] S. Mysore, B. Mazloom, B. Agrawal, and T. Sherwood, "Understanding and visualizing full systems with data flow tomography," *ACM SIGPLAN Notices*, vol. 43, no. 3, pp. 211–221, 2008.

[36] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica, "X-trace: A pervasive network tracing framework," in *Proceedings of the 4th USENIX conference on Networked systems design & implementation*, 2007, p. 20–20.

[37] B. C. Tak, C. Tang, C. Zhang, S. Govindan, B. Urgaonkar, and R. N. Chang, "vPath: precise discovery of request processing paths from black-box observations of thread and network activities," in *Proceedings of the 2009 conference on USENIX Annual technical conference*, 2009, p. 19–19.

[38] A. Chanda, A. L. Cox, and W. Zwaenepoel, "Whodunit: Transactional profiling for multi-tier applications," in *ACM SIGOPS Operating Systems Review*, 2007, vol. 41, pp. 17–30.

[39] N. Froyd, J. Mellor-Crummey, and R. Fowler, "Low-overhead call path profiling of unmodified, optimized code," in *Proceedings of the 19th annual international conference on Supercomputing*, 2005, pp. 81–90.

[40] A. Mirgorodskiy and B. Miller, "Diagnosing distributed systems with self-propelled instrumentation," *Middleware 2008*, pp. 82–103, 2008.

[41] C. Q. Yang and B. P. Miller, "Critical path analysis for the execution of parallel and distributed programs," in *8th International Conference on Distributed Computing Systems*, 1988, pp. 366–373.

[42] J. K. Hollingsworth, "An online computation of critical path profiling," in *Proceedings of the SIGMETRICS symposium on Parallel and distributed tools*, 1996, pp. 11–20.

[43] A. G. Saidi, N. L. Binkert, S. K. Reinhardt, and T. Mudge, "Full-system critical path analysis," in *IEEE International Symposium on Performance Analysis of Systems and software (ISPASS)*, 2008, pp. 63–74.

[44] G. Franks, T. Al-Omari, M. Woodside, O. Das, and S. Derisavi, "Enhanced modeling and solution of layered queueing networks," *IEEE Transactions on Software Engineering*, vol. 35, no. 2, pp. 148–161, 2009.

[45] G. Franks, D. Lau, and C. Hrischuk, "Performance measurements and modeling of a java-based session initiation protocol (SIP) application server," in *Proceedings of the joint ACM SIGSOFT conference–QoSA and ACM SIGSOFT symposium–ISARCS on Quality of software architectures–QoSA and architecting critical systems–ISARCS*, 2011, pp. 63–72.

[46] T. A. Israr, D. H. Lau, G. Franks, and M. Woodside, "Automatic generation of layered queuing software performance models from commonly available traces," in *Proceedings of the 5th international workshop on Software and performance*, 2005, pp. 147–158.

[47] F. Ricciato and W. Fleischer, "Bottleneck detection via aggregate rate analysis: a real case in a 3G Network," in *Network Operations and Management Symposium, 2006. NOMS 2006. 10th IEEE/IFIP*, 2006, pp. 1–4.

[48] "Event Tracing for Windows." [Online]. Available: http://msdn.microsoft.com/en-us/library/windows/desktop/ff190903(v=vs.85).aspx. [Accessed: 30-Mar-2012].

[49] B. M. Cantrill, M. W. Shapiro, and A. H. Leventhal, "Dynamic instrumentation of production systems," in *Proceedings of the annual conference on USENIX Annual Technical Conference*, 2004, p. 2–2.

[50] F. C. Eigler and R. Hat, "Problem solving with systemtap," in *Proceedings of the Ottawa Linux Symposium*, 2006, vol. 2006.

[51] "Red Hat Enterprise Linux Developer Guide." [Online]. Available: http://docs.redhat.com/docs/en-US/Red_Hat_Enterprise_Linux/6/html/Developer_Guide. [Accessed: 30-Mar-2012].

[52] F. Giraldeau, J. Desfossez, D. Goulet, M. Dagenais, and M. Desnoyers, "Recovering System Metrics from Kernel Trace," in *Linux Symposium*, 2011, p. 109.

[53] "Rice University Bidding System." [Online]. Available: http://rubis.ow2.org/. [Accessed: 12-Mar-2012].